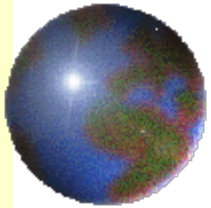
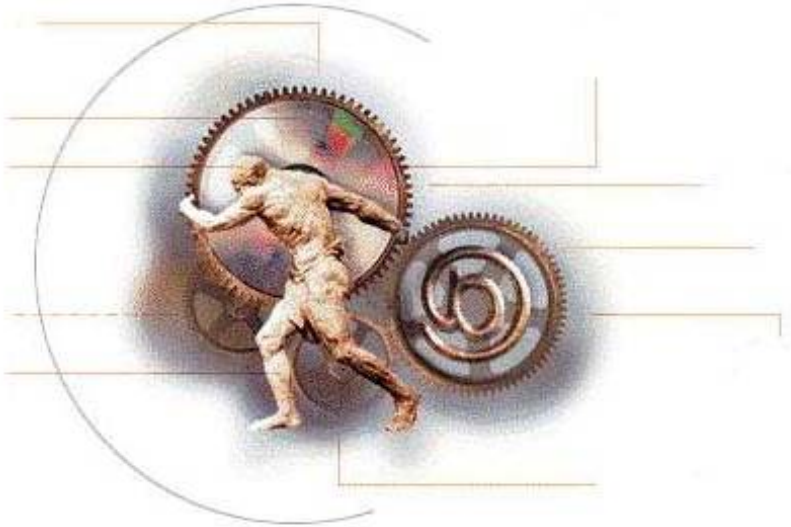


資料結構(Data Structures)

Course 1: Algorithm and Time Complexity



授課教師:陳士杰

國立聯合大學　資訊管理學系

Outlines

- 本章重點
  - Algorithm Def. (5個性質)
  - Data Type and ADT (Abstract Data Type)
  - Program Complexity
    - Space complexity (空間複雜度)
    - Time complexity (時間複雜度)
      - O, Ω, θ

Algorithm

- 演算法：**能夠利用電腦解決問題的步驟**。
- 通常在針對某一問題開發程式時，都會經過下列過程：

  Step 1: 明確**定義問題**

  Step 2: **設計演算法，並估計其執行時間與所使用之記憶體空間大小**

  Step 3: 撰寫**程式**，並加以測試

- Example: 設計一程式以計算大學入學考試中，某一單科分數之高標
  - **明確定義**：計算所有考生在該科中**前25%成績**之平均。
  - **演算法**：

    Step 1: 將所有考生該科成績**排序** (由高至低)

    Step 2: 將排名在前面1/4的成績資料相加後，再除以1/4的人數
  - **撰寫程式**：...

- Def: 完成特定功能之有限個指令之集合。同時，需滿足下列5個特性：
  - **Input** (輸入): 外界至少提供≥0個輸入
  - **Output** (輸出): Algorithm至少產生≥1個輸出結果
  - **Definiteness** (明確): 每個指令必須是Clear and Unambiguous
  - **Finiteness** (有限性): Algorithm在執行有限個步驟後，必定終止
  - **Effectiveness** (有效性): 用紙跟筆即可追蹤Algorithm中執行的過程及結果

- 其中，Program不一定會滿足特性4，但是Algorithm一定會滿足該特性!!
  - Ex: 軍事防衛系統、提款機系統、作業系統
  - 這就是Program與Algorithm不同之處

Pseudocode (虛擬碼)

- 常用的演算法表示工具有哪些?
  - 只要是能夠**闡述問題解決步驟**的工具皆可!!
    - 按照步驟次序，逐一以人類口語條列之
    - 流程圖 (Flowchart)
    - **虛擬碼 (Pseducode)**
    - 程式 (Program)
- 虛擬碼是目前設計演算法最常使用的工具。
  - 虛擬碼在陳述解題步驟時，是採用**介於人類口語與程式語法之間**的表達方式
  - 可以表示得很口語，也可以表示得很程式。通常建議介於兩者之間
  - 好處：
    - 在陳述問題的解題步驟時，能同時具有簡便性與邏輯性
    - 容易轉換成程式指令

- 問題: 循序搜尋

Example 1.2

The following is an example of a problem:

Determine whether the number x is in the list S of n numbers. The answer is yes if x is in S and no if it is not.

- 範例:

Example 1.4

An instance of the problem in Example 1.2 is

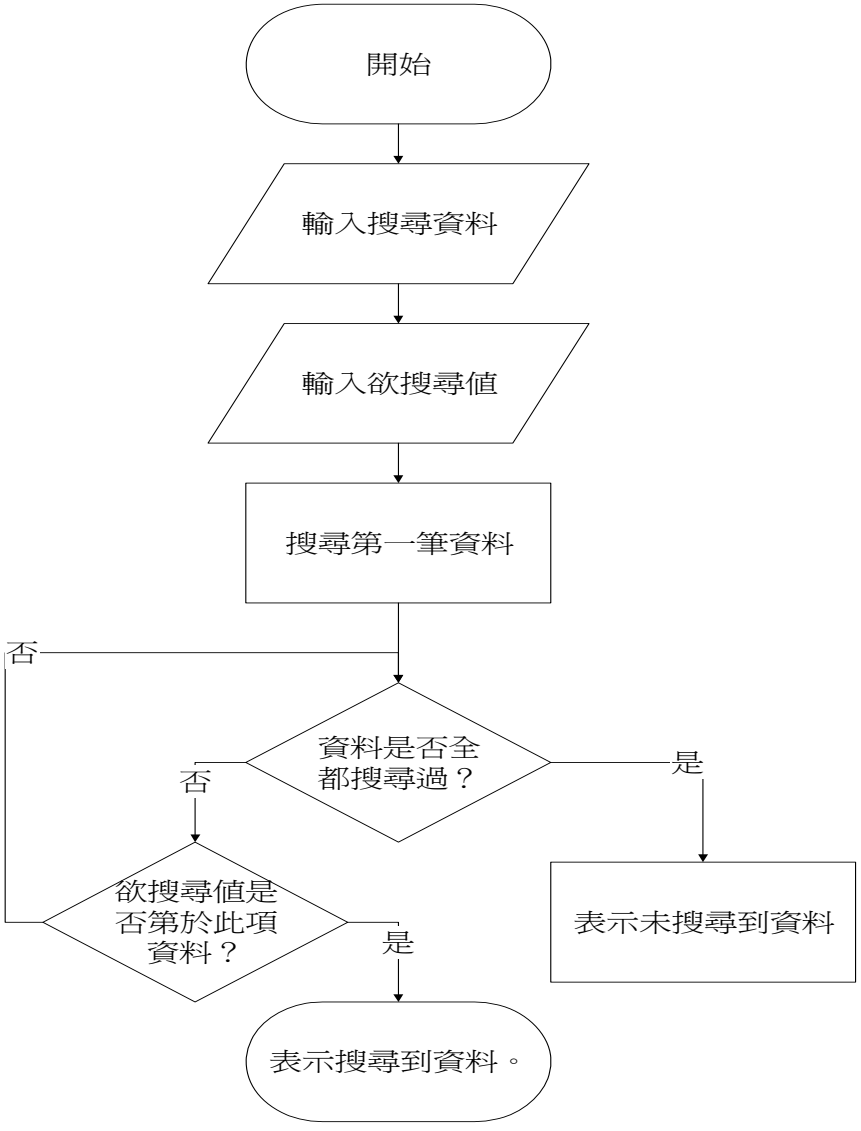
$S = [10, 7, 11, 5, 13, 8]$, $n = 6$, and $x = 5$

The solution to this instance is, "yes, x is in S."

條列式的步驟

1. 輸入資料矩陣S[ ]和欲搜尋值x。
2. 從資料矩陣S[ ]中的第一項開始搜尋。
3. 如果欲搜尋的資料x不等於目前資料矩陣中的某項資料，則搜尋下一項資料。
4. 如果欲搜尋值x與資料矩陣中的某項資料相同，則表示搜尋到資料，此時回傳 'yes'。
5. 如果資料全都搜尋過且未能搜尋到欲搜尋值，表示未能搜尋到資料，此時回傳 'not found'。

流程圖


開始
輸入搜尋資料
輸入欲搜尋值
搜尋第一筆資料
否
資料是否全都搜尋過？
否
是
欲搜尋值是否第於此項資料？
是
表示未搜尋到資料
表示搜尋到資料。

程式敘述

```
void seqsearch(int n,
               const keytype S [ ],
               keytype x,
               index & location)

{
     location = 1;
     while (location <=  n &&  S[location] != x)
       location ++;
     if (location > n)
        location=0;
}
```

演算法:

Input:正整數 n，矩陣 S[1…n]，要搜尋的值 x

Output:如果x存在於矩陣S中，則回傳yes；否則回傳not found

```
string seqsearch(int n, const S[ ], keytype x)
{
 location = 1;
 while (當尚未從矩陣S中找到所要的值，或是尚未將矩陣從頭到尾走完一趟)
        location++;
 if (location>=n)
    回傳 'not found';
 else
    回傳 'yes';
};
```

No standard for pseudocode

- 可以寫得很像**文字敘述**
- 也可以寫得很像**程式語句**

Algorithm Efficiency (演算法效能)

- 一個Algorithm的效能好壞，通常有兩個評估因子：
  - **Space (空間)**: Space Complexity 空間複雜度
  - **Time (時間)**: Time Complexity 時間複雜度
- 不論是衡量空間還是時間的複雜度，通常我們均假設演算法欲處理的資料量$n \to \infty$。這是因為：
  - 無法猜測出系統實際所面對的問題有多複雜
  - 當假設$n \to \infty$，則有一些非預期的不確定因素可以忽略。

Time Complexity

- Def:
  - 用來衡量演算法在執行時，可能所需花費的時間
  - 為衡量演算法優劣的重要依據
- **分析方法**：統計Algorithm(或Program)中，指令執行次數的總合，做為該Algorithm (或Program)的時間函數。
  - **時間函數** T(n)：表示當輸入資料量為n時，演算法中所有指令所需的實際執行次數。
  - 暫不考慮指令本身的複雜與否。

例 1.

每行指令的執行次數	程式
	float sum(float list[ ], int n)
	{
	int i;
	float tempsum = 0;
	for (i=0; i<n; i++)
	{
	tempsum += list[i];
	}
	return tempsum;
	}

不可執行!!˙·.˙就系統執行的角度而言，變數宣告只是在Compile時，於M.M.中建立一個空間，但不會產生相對應的執行碼!!除非在宣告的同時有指派一個初始值，指派的動作就會有相對應的執行碼產生以供程式執行時使用。

例 2.

每行指令的執行次數	程式
	float rsum(float list[ ], int n)
	{
	if (n>0)
	{
	return rsum(list, n-1)+ list[n-1];
	}
	return list[0];
	}

遞迴運算!!∵本題是針對list[ ]中的所有值做累加的計算，即：list[n]+ list[n-1]+ …+list[0]。

例 3.

每行指令的執行次數	程式

	Void add(int a[ ][max_size],
	int b[ ][max_size],
	int c[ ][max_size],
	int n, int m)
	{
	for (i=0; i<n; i++)
	{
	for (j=0; j<m; j++)
	{
	c[i][j] = a[i][j]+b[i][j];
	}
	}
	}

- 此for迴圈的判斷式本身會執 行n+1次，而此迴圈的程式主體會執行n次。

- 此for迴圈的判斷式本身會執行m+1次，而此迴圈的程式主體會執行m次。
- 然而，此for迴圈是在“n”這個迴圈的程式主體內，所以，此for迴圈的判斷式總共會執行(m+1)×n次

- 由上述例子可知，如果一個演算法是線性的(Linear, 即：此演算法沒有迴圈或是遞迴)，則此演算法的時間函數即為該演算法中的指令個數。
- 然而，若一個演算法有迴圈或是遞迴，則這些具重覆執行特性的語法將會主宰演算法的時間函數。
  - focuses on **迴圈**與**遞迴**.

Ex:

```
    :                for(i=1; i<=n; i++)          for(i=1; i<=n; i++){
x = x+1;             {                               for(j=1; j<=n; j++)
    :                   x = x+1;                        x=x+1;
    :                      :                               :
    :                }                            }
```

```
i = 1
loop (i <= n)
      application code
      i = i+1
end loop
```

※ T(n) = n

(application code可被執行n次)

```
i = 1
loop (i < n)
      application code
      i = i × 2
end loop
```

∵ i ⇨ 1, 2, 4, 8, 16,..., 2^k

∴ $2^k < n$ ⇨ $k < \log_2 n$

※ $T(n) = \log_2 n$

```
i = 1                                   ❶
loop (i <= n)
  ❷ j = 1
    loop (j <= n)
        application code
        j = j × 2
    end loop
    i = i + 1
end loop
```

⇨❶ $T(n_1) = n$

⇨❷ $T(n_2) = \log_2 n$

$\therefore T(n) = T(n_1) \times T(n_2)$

$= n \log_2 n$

```
i = 1                                   ❶
loop (i <= n)
  ❷ j = 1
    loop (j <= i)
        application code
        j = j + 1
    end loop
    i = i + 1
end loop
```

$T(n) = 1+2+3+4+\ldots+n$

$= n(n+1)/2$

⇨❶ $T(n_1) = n$

⇨❷ $T(n_2) = T(n)/T(n_1)$

$= (n+1)/2$

Asymptotic Notation (漸近式表示)

- 現今已較少直接使用指令的實際執行次數 (即：時間函數) 來衡量演算法的執行效率，而是採用不同的**等級 (Order)** 劃分，來對演算法所需的執行時間做分級的工作。此理念即為漸近式表示 (Asymptotic Notation)。
- 主要原因：
  - 有些程式間**實際執行次數看似不同，但是在電腦上執行的效能卻差不多**。

for i=1 to n do
  a=(b+c)/d+e;
	for i=1 to n do
	  x1=b+c;
	  x2=x1/d;
	  a=x2+e;

⇨T(n)=n	⇨T(n)=3n

  - **指令有的簡單，有的複雜**。如：
    - 浮點數運算比整數運算難
    - 除法和加法運算的複雜性不同

- Asymptotic Notation共有三種表示方法:
  - **Big-O (O)**
  - **Omega (Ω)**
  - **Theta (θ)**

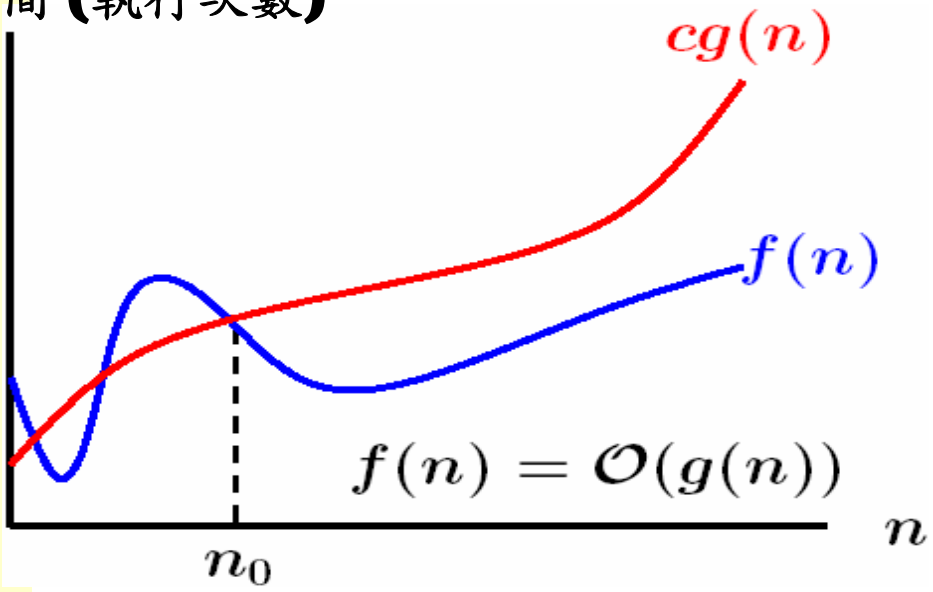
Big-O (O)

- 某演算法時間函數的**上限 (Upper bound)**
  - 即:演算法在執行時所花費的時間成長，最差的情況不會超過它
- 通常，一個時間函數的Big-O notation 能夠由以下兩個步驟所導出:
  - In each term, set the coefficient of the term to **1**.
  - **Keep the largest term** in the function and discard the others.
- 例如，有以下兩個不同的時間函數f(n):
  - $f(n) = 3n+2$, 則 $f(n) = O(n)$.
  - $f(n) = 5n^2+3n+2$, 則 $f(n) = O(n^2)$.

Definition:

- f(n) = O(g(n)) if and only if 存在兩正數c和n_o, 使得$f(n) \le c \times g(n)$, for all $n \ge n_o$.

時間 (執行次數)



- n:輸入資料量大小。
- f(n):在理想狀況下，程式在電腦中的**指令實際執行次數**。
- g(n):執行時間的**成長率**。

- Definition說明:
  - 只要n大到某一個程度 (n_0)，就保証時間函數 f(n) 的數值一定小於等於 g(n) 函數。
  - 亦即：在最壞 (worst case) 的情況下，該演算法的時間函數f(n)之成長最多會到達g(n)，而不會超過它!!

- $f(n) = 3n+2$, 則 $f(n) = O(n)$.
  - $f(n) = O(n)$ if and only if 存在兩正數c =__和n_o =__, 使得$f(n) \le c \times g(n)$, for all $n \ge n_o$.
  - 先決定c的值，鎖定f(n)中的最大項之值(即: 3n)，c只要**比該項的常數值大1**即可!!再由c去推n_o。
  - $3n+2 \le 4n \Rightarrow n \ge 2$。
- $f(n) = 5n^2+3n+2$, 則 $f(n) = O(n^2)$.
  - $f(n) = O(n^2)$ if and only if 存在兩正數c =__和n_o = __, 使得$f(n) \le c \times g(n)$, for all $n \ge n_o$.
  - 先決定c的值，鎖定f(n)中的最大項之值(即: $5n^2$)，c只要**比該項的常數值大1**即可!!再由c去推n_o。
  - $5n^2+3n+2 \le 6n^2 \Rightarrow 3n+2 \le n^2 \Rightarrow$ 取 $n_o = 4$。

Theorem: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof:

$$f(n) = \sum_{i=0}^{m} a_i \times n^i \le \sum_{i=0}^{m} |a_i| \times n^i$$

最大項

$$f(n) \le (|a_0| \times n^0 + |a_1| \times n^1 + \ldots + |a_{m-1}| \times n^{m-1} + |a_m| \times n^m)$$

$$= n^m (|a_0| \times n^{-m} + |a_1| \times n^{1-m} + \ldots + |a_{m-1}| \times n^{-1} + |a_m| \times 1)$$

$$= n^m \sum_{i=0}^{m} |a_i| \times n^{i-m}$$

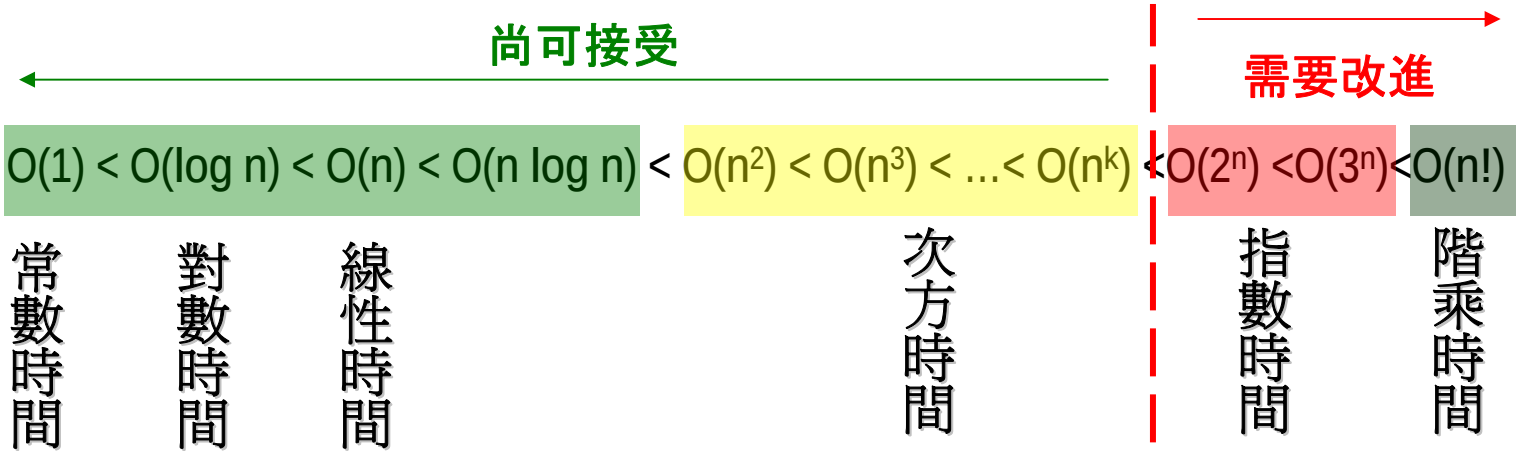
$$\le n^m \sum_{i=0}^{m} |a_i| \times 1 = n^m \sum_{i=0}^{m} |a_i| , \quad \text{for } n \ge 1$$

存在兩正數$c = \sum_{i=0}^{m} |a_i|$和$n_o =$ ___, 使得$f(n) \le c \times g(n)$, for all $n \ge n_o$.

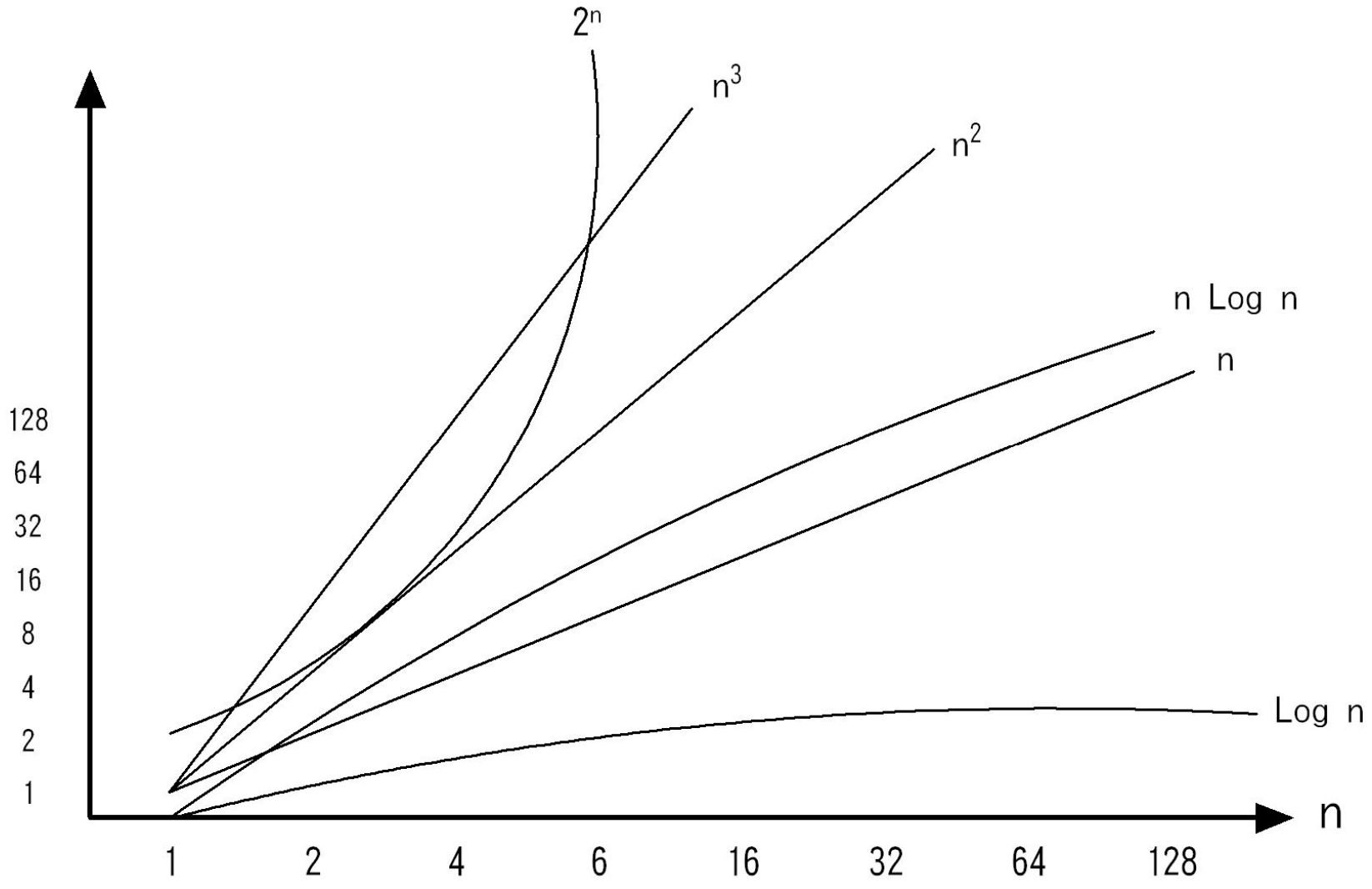
∵

⇨ So, **$f(n) = O(n^m)$**

- 一般常見演算法的計算時之Order 大小排序如下：



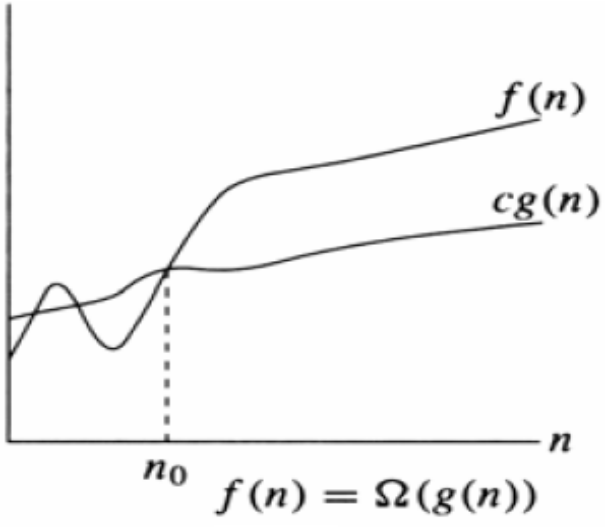
- 對於較大的資料集合(如: 生物資訊議題之資料量)，若演算法的複雜度超過**O(n log n)**時，通常都會很難處理，因為所需執行的步驟太多了。


2^n
n^3
n^2
n Log n
n
Log n
128
64
32
16
8
4
2
1
n
1
2
4
6
16
32
64
128

- 前面所提例子中的$f(n) = 3n+2$, 其Big-O為 $O(n)$。然而, $O(n^2)$, $O(n^3)$, 乃至$O(2^n)$也都是$3n+2$的上限值。
- 同理, $f(n) = 5n^2+3n+2$ 的Big-O 為 $O(n^2)$, 也可以為$O(n^3)$, 乃至$O(2^n)$, 這是因為:
  - $f(n) = O(g(n))$這個式子只說明了當$n \geq n_0$時, 若$f(n) \leq c \times g(n)$, 則$g(n)$是$f(n)$的上限值, 但**並不能看出該上限值是多高**!!
  - 但是, 為了要使$f(n) = O(g(n))$這個式子更有意義, $g(n)$必須要儘量**小**。
  - 所以, $3n+2$的Big-O應為 $O(n)$, 不說 $3n+2 = O(n^3)$..., 即使這些式子也是對的。

Omega (Ω)

- 某演算法時間函數的**下限 (Lower bound)**
  - 即:演算法在執行時所花費的時間成長，最好的情況也不會低於它
- **Definition:**
  - f(n) = Ω(g(n)) if and only if 存在兩正數c和n_o, 使得f(n) ≥ c×g(n), for all n ≥ n_o.



lower bound

- Definition說明:
  - 只要n大到某一個程度 (n_0)，就保証 f(n) 函數的數值一定大於等於 g(n) 函數。
  - 亦即在最好最好 (best case) 的情況下，該演算法的時間函數f(n)之成長只能到達g(n)，不會再低於它!!

- $f(n) = 3n+2$, 則 $f(n) = \Omega(n)$.
  - $f(n) = \Omega(n)$ if and only if 存在兩正數c =__和n_o =__, 使得$f(n) \geq c \times g(n)$, for all $n \geq n_o$.
  - 先決定c的值，鎖定f(n)中的最大項之值(即: 3n)，c只要**與該項的常數值相同**即可!!再由c去推n_o。
  - $3n+2 \geq 3n \Rightarrow 2 \geq 0$ (恒真，故n_o可任取!)。
- $f(n) = 5n^2+3n-2$, 則 $f(n) = \Omega(n^2)$.
  - $f(n) = \Omega(n^2)$ if and only if 存在兩正數c =__和n_o =____, 使得$f(n) \geq c \times g(n)$, for all $n \geq n_o$.
  - 先決定c的值，鎖定f(n)中的最大項之值(即: $5n^2$)，c只要**與該項的常數值相同**即可!!再由c去推n_o。
  - $5n^2+3n-2 \geq 5n^2 \Rightarrow 3n-2 \geq 0 \Rightarrow$ 取 $n_o = 2/3$。

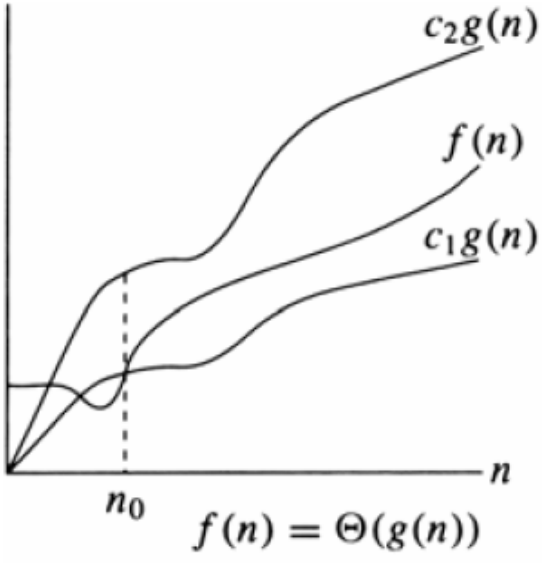
- 前面所提例子中的f(n) = 3n+2, 其Omega為 $\Omega(n)$, 然而, $\Omega(1)$也是3n+2的下限值; 同理, $f(n) = 5n^2+3n+2$ 的Omega為 $\Omega(n^2)$, 也可以為$\Omega(n)$, 乃至$\Omega(1)$. 這是因為:
  - f(n) = Ω(g(n))這個式子只說明了當$n \geq n_0$時, 若**f(n) ≥ c×g(n)**, 則g(n)是f(n)的下限值, 但並不能看出該下限值是多低!!
  - 但是, 為了要使f(n) = Ω(g(n))這個式子更有意義, g(n)必須要儘量**大**。
  - 所以, 3n+2的Omega應為 $\Omega(n)$, 不說 3n+2 = $\Omega(1)$…, 即使這個式子也是對的。

Theta (θ)

- More precise than O and Ω .
- **Definition:**
  - f(n) = θ(g(n)) if and only if 存在三正數c_1, c_2和n_o, 使得

$$c_1 \times g(n) \le f(n) \le c_2 \times g(n),$$

for all $n \ge n_o$.



tight bound

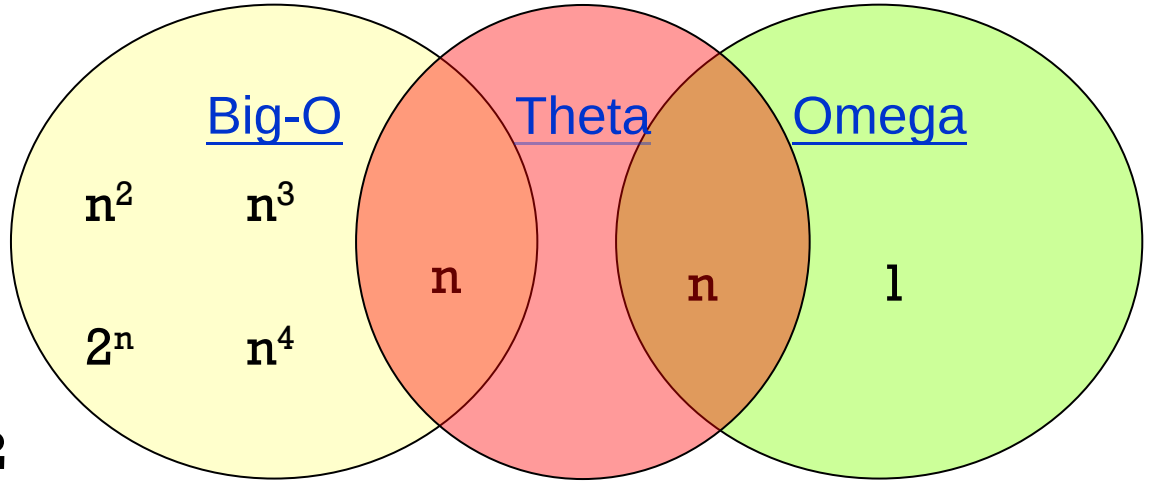
- Definition說明:
  - 在夠大的n值時，如果存在有正的常數c_1與c_2，來讓f(n)夾於c_1 g(n)與c_2 g(n)之間，則f(n)即屬於θ(g(n)) 之集合。

- $f(n) = 3n+2$, 則 $f(n) = \theta(n)$.
  - $f(n) = \theta(n)$ if and only if 存在三正數c_1 =__, c_2 =__和n_o =__, 使得 $f(n) \geq c \times g(n)$, for all $n \geq n_o$.
  - 用先前決定O 與 Ω中c值的方式分別得到c_1 與 c_2 即可!!再由c_1 與 c_2去推n_o。
  - $3n+2 \leq 4n$ (用$f(n) \leq c_2 \times g(n)$來找) ⇨ $n \geq 2$ 。

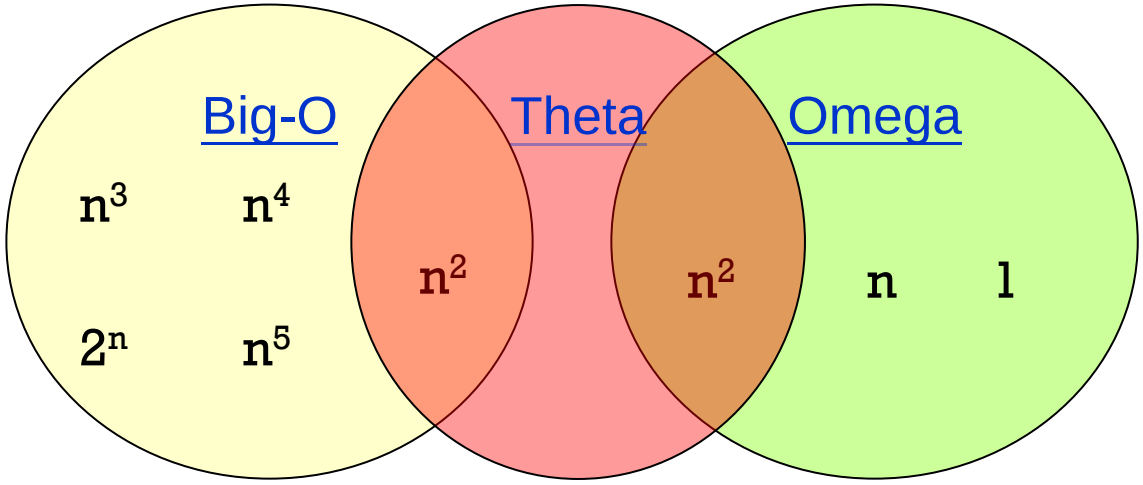
※ If $f(n) = O(g(n))$且$f(n) = \Omega(g(n))$, 則**$f(n) = \theta(g(n))$**.

Big-O, Omega與Theta的關係

- 以f(n) = 3n+2 與 f(n) = $5n^2+3n+2$為例:
  - 3n+2:



  - $5n^2+3n+2$



※練習範例※

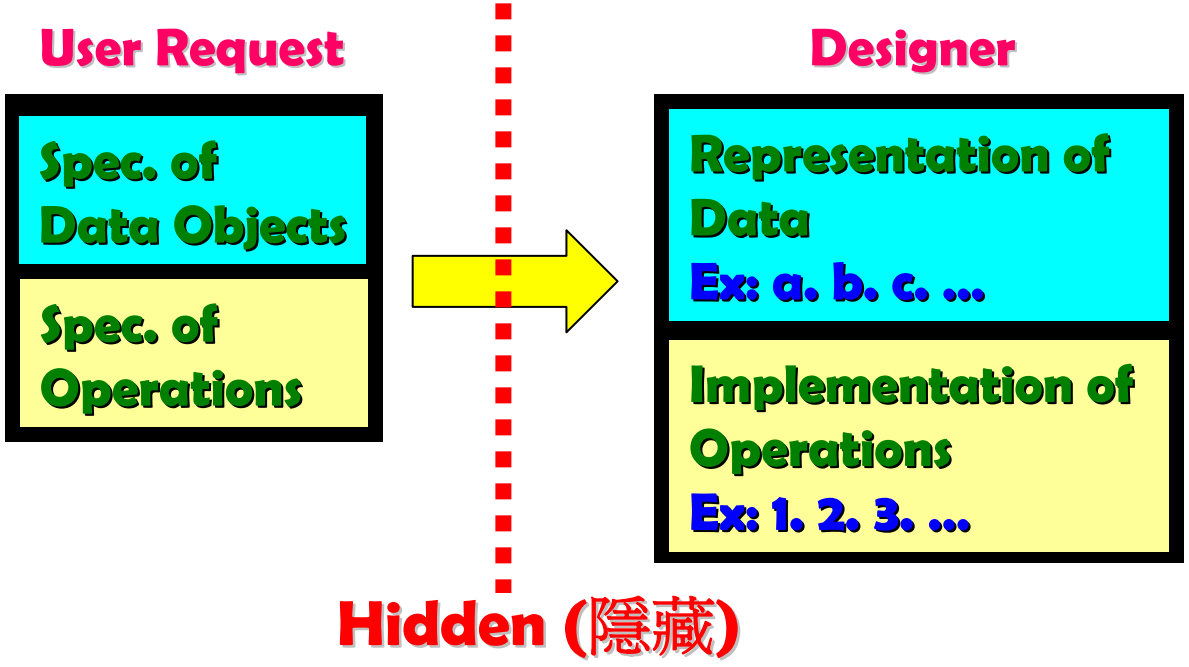
- 試說明下列等式是正確的:
  - $5n^2 - 6n = \theta(n^2)$
  - $100n+n\log n+500 = O(n\log n)$
  - $33n^3+4n^2 = \Omega(n^2)$

Data Type and Abstract Data Type

- Data Type (資料型態)
  - Def: A data type consists of two parts
    - a set of **data**
    - the **operations** that can be performed on the data.
- For example:
  - Integer
    - Data: -∞, …, -2, -1, 0, 1, …, ∞ (沒有小數的數值集合)
    - Operations: +, -, ×, ÷, %, ≤, ≥, ==, !=, ++, --, …
  - Floating point
    - Data: -∞, …, -1.9, …, 0.0, …, ∞
    - Operations: +, -, ×, ÷, %, ≤, ≥, ==, !=, …
  - Character
    - Data: ‘A’, ‘B’, …, ‘a’, ‘b’, …
    - Operations: <, >, …

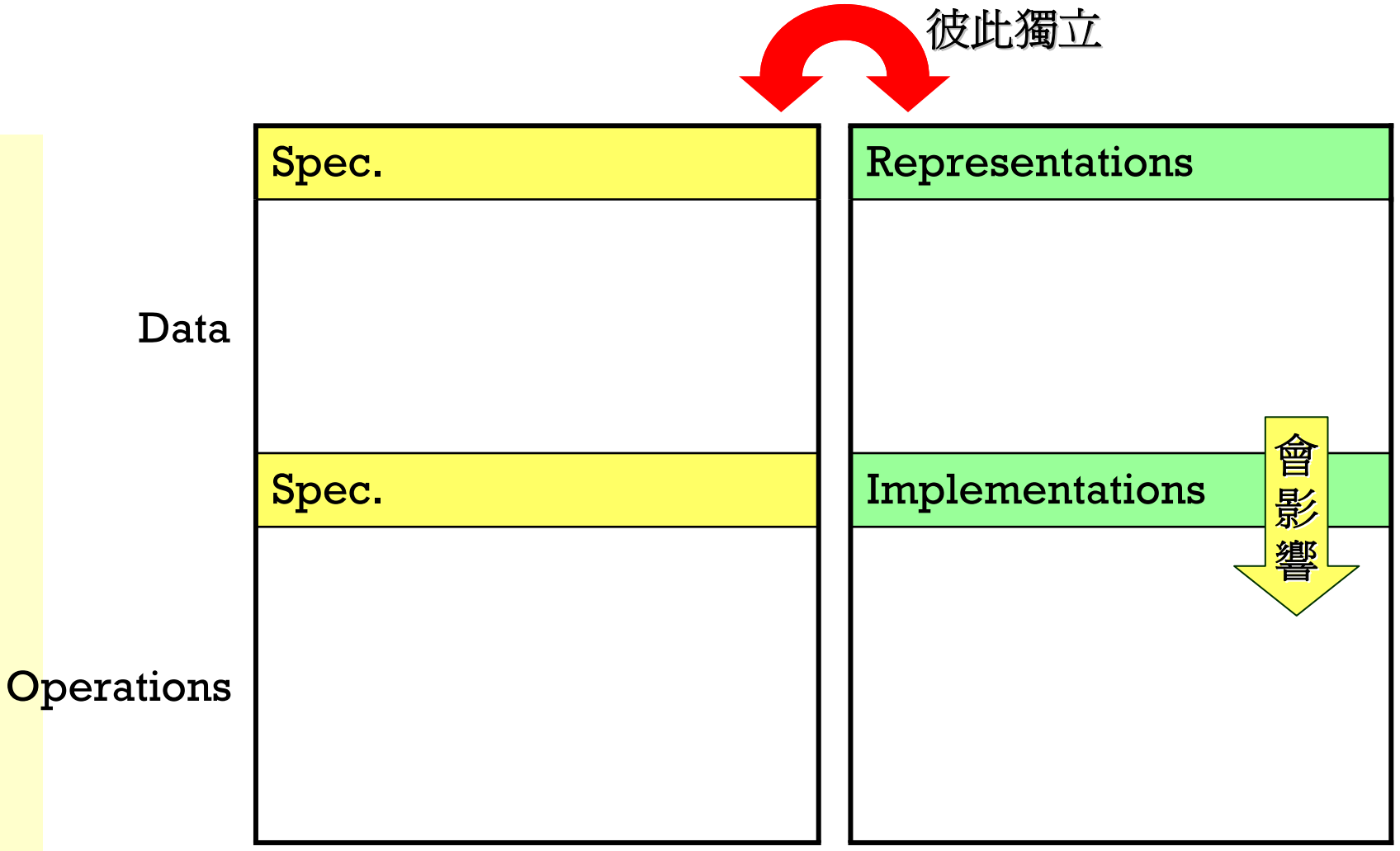
ADT (Abstract Data Type; 抽象資料型態)

- **Def**: ADT是一種Data Type，且要滿足："**The specification (spec.) of data** and **the spec. of operations**" are independent with "**the representation of data** and **the implementation of operations.**"



- 此定義隱含了兩個意義:
  - Spec.不會和Implementation掛在一起
  - 任何Implementation的改變，都不會影響到該ADT的定義和外界User對它的使用方式

例：Stack (堆疊)的ADT



補　充

Space Complexity

- 空間複雜度(**S(P)**)通常來自於兩方面:
  - **Fixed Space Requirement**: **C**
    - Instruction Space (即:程式碼大小)
    - 變數 (**Simple variables**, 如:int, float, ...)
    - Constant
    - Fixed size **structure variables** (如:陣列、紀錄...等, 用**struct**或**class**來宣告)
  - **Variable Space Requirement**: **SP(I)**
    - 參數: 若**structure variables**是以**call by value**為主的之參數傳遞方式 (如:陣列...)
    - 由於Recursive Call所需要的**Stack空間**

非主要考量
(∵C這個值在程式設計完成後就固定了)

主要考量
(∵S(P)會與SP(I)呈線性關係)

⇨ **S(P) = C + SP(I)** [C是固定常數, **SP(I)**是會變動的變數]

型態種類	資料型態	佔記憶體空間	範圍
整數型態	int	4 Bytes	-2,147,483,648 ~ 2,147,483,647
	short (short int)	2 Bytes	-32,768 ~ 32,767
	long (long int)	4 Bytes	-2,147,483,648 ~ 2,147,483,647
	unsigned (unsigned int)	4 Bytes	0 ~ 4,294,967,295
	unsigned short	2 Bytes	0 ~ 65,535
	unsigned long	4 Bytes	0 ~ 4,294,967,295
浮點數型態	float (單準確度浮點數)	4 Bytes	10^{-38} ~ 10^{38} 六位精確度
	double (倍準確度浮點數)	8 Bytes	10^{-308} ~ 10^{308} 十五位精確度
字元型態	char	1 Byte	-128 ~ 127
	unsigned char	1 Byte	0 ~ 255
資料型態前加上 unsigned 表使用無號（無正負號）資料型態 在資料型態一欄如有括號，表示資料型態也可寫成如括號內所示			

例 1.

Simple variables

```
float abc(float a, float b, float c)
   {
        return a+b+b*c(a+b-c)/(a+b)+4;
   }
```

- SP(I) = 0 (∵沒有structure variable, 也沒有Recursive Call)

例 2.

```
float sum(float list[ ], int n)
{
    float tempsum = 0;
    int i;

    for(i=0; i<n; i++) tempsum += list[i];
    return tempsum;
}
```

- SP(I)
  - 有無Stack空間花費 ⇨沒有 (∵沒有Recursive Call)
  - 有structure variable，考量參數傳遞是不是call by value:
    - = **4×n**, list[ ]若為**call by value** 傳遞 (根據主程式所傳來的**數值多寡**)
    - = **0 (或一常數)**, list[ ]若為**call by address** 傳遞 (∵主程式只傳陣列的**起始位址**，沒有變動空間需求)

例 3.

```
float rsum(float list[ ], int n)
{
    if(n!=0) return rsum(list, n-1)+list[n-1];
    return list[0];
}
```

- 假設: int 佔4 bytes, float佔4 bytes, Address佔2 bytes, list[ ]以call by address傳遞
- SP(I)
  - 有structure variable，但參數傳遞方式不是call by value ⇨ ∴沒有變動空間需求
  - 有無Stack空間花費 ⇨有 (∵有Recursive Call)
- **發生一次遞迴**所須的Stack空間為:

  SP(I)=(參數"list"之起始位址+參數"n")+返回位址 = (2+4)+2 = 8 bytes
- 共有**n次**Recursive call ⇨ ∴ SP(I) = **8n bytes**.

Time Complexity 相關議題

- 計算某一指令 (ex: x=x+1) 的執行次數或Big-O
- 遞迴演算法的時間函數 **(演算法課程)**

給程式片段，統計loop內執行次數及Big-O

- 例 1:

```
For i = 1 to n do
    For j = 1 to i do
        x = x+1
    end
end
```

求x=x+1之執行次數與Big-O.

Sol:

i值	i = 1	i = 2	…	i = n
j值	j = 1 to 1	j = 1 to 2	…	j = 1 to n
x=x+1	執行1次	執行2次	…	執行n次

⇨執行次數:

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

⇨ $O(n^2)$

- 例 2:

```
i = n; (n>0)
while (i >0) do
      x = x+1;
       i = i/2;
end
```

求x=x+1之執行次數與Big-O.

Sol:

i = n
= n/2 $\Leftarrow n/2^1$
= n/4 $\Leftarrow n/2^2$
= …
= $n/2^k = 1 \Rightarrow 2^k = n$

⇨執行次數: $\log_2 n$

⇨ $O(\log_2 n)$

- 例 3:

```
For i = 1 to n do
    For j = 1 to i do
        For k = 1 to j do
            x = x+1;
        end;
    end;
end
```

求x=x+1之執行次數與Big-O.

Sol:

i值	i = 1	i = 2		i = 3			…	i = n			
j值	j = 1 to 1	j = 1 to 2		j = 1 to 3			…	j = 1 to n			
k值	k = 1 to 1	k = 1 to 1	k = 1 to 2	k = 1 to 1	k = 1 to 2	k = 1 to 3	…	k = 1 to 1	k = 1 to 2	…	k = 1 to n
x=x+1	執行1次	執行1次	執行2次	執行1次	執行2次	執行3次	…	執行1次	執行2次	…	執行n次

共1次　共3次　共6次　共(1+2+…+n)次

⇨n(n+1)/2

⇨執行次數:

$$\sum_{i=1}^{n}\frac{i(i+1)}{2}=\frac{1}{2}\left[\sum_{i=1}^{n}i^2+\sum_{i=1}^{n}i\right]=\frac{n(n+1)(n+2)}{6}$$

⇨ $O(n^3)$

- 例 4:

```
For k = 1 to n do
    For i = 1 to k do
        For j = 1 to k do
            if (i ≠ j) then x=x+1;
        end;
    end;
end
```

求x=x+1之執行次數與Big-O.

(Hint: k^2 – (i = j的次數))

Sol:

⇨執行次數:

i = j 的情況

$$\sum_{k=1}^{n}(k^2-k)=\sum_{k=1}^{n}k^2-\sum_{k=1}^{n}k$$

$$=\frac{n(n+1)(2n+1)}{6}-\frac{n(n+1)}{2}$$

$$=\frac{n(n+1)(n-1)}{3}$$

⇨ $O(n^3)$

常用的數學式子

(1) 等差數列：$\sum_{i=1}^{n} i = 1+2+3+\ldots+n = \frac{n(n+1)}{2}$

(2) 等比數列：$\sum_{i=0}^{n} r^i = r^0 + r^1 + r^2 + \ldots + r^n = \frac{r^{n+1} - r^0}{r-1}$

(3) $C_m^n = \frac{n!}{m!(n-m)!}$, $P_m^n = \frac{n!}{m!}$

(4) $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$

① $\log \log n = \log (\log n)$

② $\log^k n = (\log n)^k$

③ $a = b^{\log_b a}$

④ $\log_c ab = \log_c a + \log_c b$

⑤ $\log_b a^n = n \log_b a$

⑥ $\log_b a = \log_c a / \log_c b$

⑦ $\log_b a = 1 / \log_a b$

⑧ $\log_b (1/a) = \log_b a^{-1} = -\log_b a$

⑨ $a^{\log_b c} = c^{\log_b a}$